|QuantumPath⟩®
Tutorials

Implementation of Simon's algorithm in qSOA®

© QuantumPath® 2022 | All rights reserved

Contents

1 Simon's algorithm **2**
1.1 Previous concepts 2
1.2 The problem 2
1.3 Classical solution 2
1.4 Quantum solution 3
1.5 Usefulness of the algorithm 5

2 Implementation of the algorithm in qSOA® **5**
2.1 Setting up qSOA® 5
2.1.1 Securing the connection 6
2.2 Assigning a circuit to the solution 6
2.2.1 Defining the circuit 6
2.2.2 Creating the circuit 7
2.3 Assigning a circuit flow to the circuit 7
2.3.1 Defining the flow 8
2.3.2 Creating the flow 8
2.4 Implementation of the algorithm 9
2.4.1 Executing the algorithm 9
2.4.2 Multiple devices 11

© QuantumPath® 2022 | All rights reserved

1 Simon's algorithm

1.1 Previous concepts

Let's define some mathematical concepts that are going to be used throught the tutorial.

- XOR gate: It is a digital logic gate that gives a true output when the number of true inputs is odd, - in other words, it produces a 0 when both inputs match. For two given inputs, A and B, the XOR gate is represented by the expression: $A \oplus B$. The XOR gate is equivalent to an addition modulo-2. As an example, we can compute $100110 \oplus 001101 = 101011$. These are the properties of the XOR gate:
  - Commutativity: $A \oplus B = B \oplus A$
  - Associativity: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
  - Identity element: $A \oplus 0 = A$
  - Self-inverse: $A \oplus A = 0$
- One-to-one function: Maps one unique input for every output. For example: $f(1) \rightarrow 1$ $f(2) \rightarrow 2$ $f(3) \rightarrow 3$ $f(4) \rightarrow 4$
- Two-to-one function: Maps two inputs for every output. For example: $f(1) \rightarrow 1$ $f(2) \rightarrow 2$ $f(3) \rightarrow 1$ $f(4) \rightarrow 2$

1.2 The problem

Given an unknown black-box function, $f : \{0,1\}^n \rightarrow \{0,1\}^n$, we know that for some unknown numbers, s, $f(x) = f(x \oplus s)$. Noticing that for two binary numbers $x_1 \neq x_2$ then $x_1 \oplus s \neq x_2 \oplus s$, so if $s = 0^n$ then f is a one-to-one function. However, if $s \neq 0^n$ then f is a two-to-one function. The problem consists of finding the number s making as few queries as possible.

1.3 Classical solution

One way of finding the classical solution is by giving values to the function and seeing which outputs we obtain - the reader must remember that the function f is supposed to be unknown to us. In this case, if the function happens to be one-to-one, then $2^{n-1} + 1$ queries will be needed to know that $s = 0^n$ with 100% probability. On the other hand, if the function is two-to-one, then we may find two inputs mapped to the same input on the first two queries, and therefore, find the number s. Or we may be unlucky and find all the different outputs on the first 2^{n-1} tries, needing then $2^{n-1} + 1$ to find the solution. We will see now how we can use quantum computing to use fewer queries.

© QuantumPath® 2022 | All rights reserved

1.4 Quantum solution

General idea and scope of the algorithm

The idea behind Simon's algorithm relies on making a circuit such that, when we run it enough times, we can use the results to find $n-1$ linearly independent n-bit strings $z_1, z_2, ...z_{n-1} \in \{0,1\}^n$ such that the following equations are satisfied:

$$\begin{aligned} z_1 \cdot s &= 0 \\ z_2 \cdot s &= 0 \\ &\vdots \\ Z_{n-1} \cdot s &= 0 \end{aligned} \tag{1}$$

Where the dot represents the modulo-2 product so that $z_i \oplus s = z_{i1}s_1 \oplus z_{i2}s_2 \oplus ... \oplus z_{in}s_n$ where $s_j \in \{0,1\}$. We note that the system has $n-1$ equations and n unknowns (the bits of $s \in \{0,1\}^n$), so the final solution may not be unique. This is because the string $00...0_n$ produces a trivial equality, as we will see afterwards.

Simon's algorithm

In figure 1, we can see a circuit implementing Simon's algorithm in QPath® for the case of a two-to-one function. We have chosen the string $s = 101$. Also, we can set arbitrary values to the image of the function with the only requirement that $f(x) = f(x \oplus s)$.

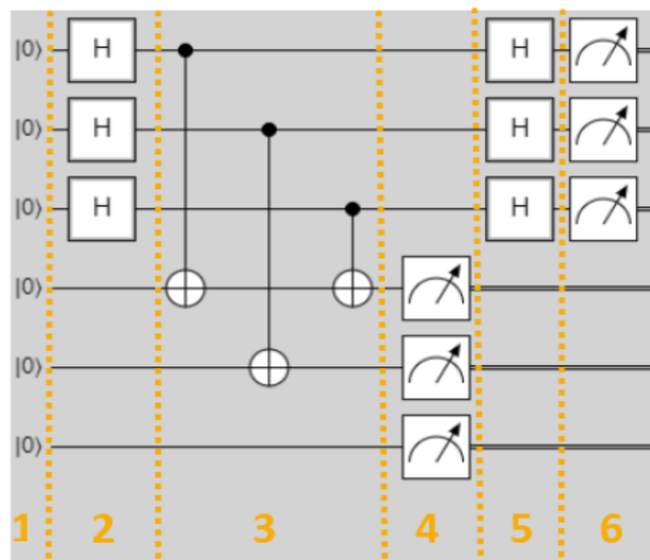


Figure 1: Simon's algorithm implemented in QPath® for s=101

It is important to notice that the quantum algorithm will be different depending on the chosen

© QuantumPath® 2022 | All rights reserved

values for the image of f. In this case, we have chosen the following function:

$$\begin{aligned} f(000) = f(101) = 000 \\ f(001) = f(100) = 100 \\ f(010) = f(111) = 010 \\ f(011) = f(110) = 110 \end{aligned} \tag{2}$$

The numbers in figure 1 correspond to the different steps of the algorithm, which will be explained now:

1. We use two quantum registers, and we can see that the algorithm starts with the input $|0^n\rangle \otimes |0^n\rangle$. We will follow the original formulation of the problem, which says that $f : \{0,1\}^n \to \{0,1\}^n$ and therefore we will also need n qubits in the second register.

2. We apply Hadamard transformations to the qubits on the first register, so that the composite state becomes: $|\Psi\rangle = (H^{\otimes n}|0^n\rangle) \otimes |0^n\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (|x\rangle \otimes |0^n\rangle)$, where $x \in \{0,1\}^n$ denotes any n-bit string.

3. In this step, we will have to implement the oracle. The gates used in the oracle will depend on the function that we want to implement. The transformation should be such that:

$$|\Psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (|x\rangle \otimes |0^n\rangle) \to \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (|x\rangle \otimes |f(x)\rangle) \tag{3}$$

   This means that the input of the function is the result of the superposition generated by the Hadamard gates on each qubit of the first register, and the output is stored on the second register as a result of applying CNOT gates, having their control qubit on the first register. This allows us to build our two-to-one function, considering the string s. For example, when the input produced by the first Hadamard gates on the first register is the string $x_1 = 000$, it will produce an output $f(x_1) = 000$ in the second register, as no control gate is being activated.

4. Now we proceed to measure the qubits in the second register, getting some value for $f(x)$. This can correspond to two different inputs: x_1 and $x_2 = x_1 \oplus s$. As the value of $f(x)$ is irrelevant, we can just focus on the state of the first register, which becomes $|\Psi'\rangle = \frac{1}{\sqrt{2}}(|x_1\rangle + |x_2\rangle)$.

5. After applying a Hadamard gate on the first register, the quantum state will become $|\Psi''\rangle = \frac{1}{\sqrt{2^{n1}}} \sum_{z\in\{0,1\}^n} [(-1)^{x_1\cdot z} + (-1)^{x_2\cdot z}] |z\rangle$.

6. When we apply a measurement to the qubits in the first register we will have an output only if $(-1)^{x_1\cdot z} = (-1)^{x_2\cdot z}$ which implies that $x_1 \cdot z = x_2 \cdot z \to x_1 \cdot z = (x_1 \oplus s) \cdot z \to x_1 \cdot z = x_1 \cdot z \oplus s \cdot z \to s \cdot z = 0(mod2)$, where the transition from the second to the third state can be shown computing a truth table. So, we will have to solve the following equation system:

$$\begin{aligned} z_1{\cdot}s = 0 \\ z_2{\cdot}s = 0 \\ \vdots \\ z_n{\cdot}s = 0 \end{aligned} \tag{4}$$

© QuantumPath® 2022 | All rights reserved

And once we have run the algorithm enough times to get independent equations, we can get the result for the string s.

1.5 Usefulness of the algorithm

Given that the secret string needs to be known to build the circuit, it is clear to see that this algorithm does not have any real-world version. However, the fact that it can solve a problem for any vector size in a single query shows us the potential of quantum computing and the exponential speed-up that it could bring to some tasks.

2 Implementation of the algorithm in qSOA®

Once we are aware of how the algorithm works, we are ready to implement it in qSOA®. This will allow us to create the circuit and execute it in different quantum computing providers, among many other things.

The process of implementing an algorithm in qSOA® is comprised of four steps:

1. Setting up qSOA® and selecting the quantum solution
2. Creating a circuit with the algorithm and assigning it to the solution
3. Introducing a circuit flow to control the number of launches of the algorithm
4. Executing the flow on different quantum devices.

As can be seen in qSOA's® manual, ref.[1], there are multiple ways to secure the connection depending on the context. Following the best practices of qSOA®, in this tutorial we are integrating the security in the code. Similarly, one can work with asynchronous or synchronous programming. Keeping in mind the purpose of this tutorial we will use the synchronous version.

2.1 Setting up qSOA®

Firstly, we import the SDK that has been previously installed, see ref.[1], and create the qSOA® workspace to work with.

[1]:
```
from QuantumPathQSOAPySDK import QSOAPlatform  # Import SDK
```

[2]:
```
# Create qSOA workspace, login manually
qsoa = QSOAPlatform()

username = 'username'
password = 'password' # password encrypted in SHA-256

authenticated = qsoa.authenticateEx(username, password)

print('Authentication completed:', authenticated)
```

© QuantumPath® 2022 | All rights reserved

```
Authentication completed: True
```

Then we review the existing solutions and select the one we are interested in.

```
[3]: # Get catalogs
solutionList = qsoa.getQuantumSolutionList()
print("    ",solutionList)
idSolution = int(input("Select idSolution: "))
```

```
    {'12345': 'QS_GateTutorials'}
Select idSolution: 12345
```

2.1.1 Securing the connection

As has been said, qSOA® allows multiple business development contexts. Therefore, the user can secure the connection through a configuration file, *.qpath*, useful at a personal level, or in parameterized way, as it is done here.

2.2 Assigning a circuit to the solution

Once the solution has been selected, we must link it to the circuit that we are interested in implementing. Therefore, we first need to define the circuit and create it.

2.2.1 Defining the circuit

As we are working with a gates circuit, we can either enter the circuit in *visual language*, VL, or *intermediate language*, IL. In this case we are going to work with VL and, therefore, we can either enter the circuit as a *String* or as a *CircuitGates* object, which is what we are going to go for.

In order to define the circuit shown in figure 1, we just need to write the gates we want to use, in the correspondent qubits.

```
[4]: ## Defining circuit w/ Circuit Gates
circuitG = qsoa.CircuitGates()
circuitG.h(list(range(3)))
circuitG.cx(0,3)
circuitG.cx(1,4)
circuitG.cx(2,3)
circuitG.h(list(range(3)))
circuitG.measure(list(range(6)))

print(circuitG.getCircuitBody())
```

```
[['H', 'H', 'H'], ['CTRL', 1, 1, 'X'], [1, 'CTRL', 1, 1, 'X'], [1, 1,
'CTRL', 'X'], ['H', 'H', 'H'], ['Measure'], [1, 'Measure'], [1, 1,
'Measure'], [1, 1, 1, 'Measure'], [1, 1, 1, 1, 'Measure'], [1, 1, 1, 1, 1,
'Measure']]
```

© QuantumPath® 2022 | All rights reserved

Note: Every gate can be applied to a single qubit, a list of them, introduced as a list, or to every qubit in the circuit, using ().

2.2.2 Creating the circuit

In order to create the circuit we are going to use the *createAssetSync* function. This function receives the following fields as inputs:

- idSolution: to associate the circuit with the solution we have selected before.
- assetName: to set the name of the circuit.
- assetNamespace: to associate the circuit with a class of circuits that share something in common. In this case, we associate the circuit with a set of basic circuits.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the circuit we have previously defined.
- assetType: to select if we are working with a gates circuit or an annealing one.
- assetLevel: to select either visual language or intermediate language, according to the definition of the circuit.

[5]:
```
## Circuit creation
assetName = 'QC_qSOA_Simon'
assetNamespace = 'Manual.Gates.Simon'
assetDescription = 'Creating Simons circuit from qSOA'

assetBody = circuitG
assetType = 'GATES'
assetLevel = 'VL'

CircuitManagementResult = qsoa.createAssetSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetType, assetLevel)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAsset* function creates, compiles and transpiles the asset, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAsset*, and the *getAssetManagementResult* function.

2.3 Assigning a circuit flow to the circuit

The quantum flow is a box diagram that will allow us to control the number of runs of our algorithm. This is very useful, as the results obtained in quantum computing have a certain probability associated with them. That means that the more times we run the algorithm, the more robust the results will be.

© QuantumPath® 2022 | All rights reserved

2.3.1 Defining the flow

As happens with the circuit, the flow can be written in VL or IL. If we choose to do it with VL then it can be entered as a *String* or a *CircuitFlow* object. Otherwise, it can only be entered as a *String*. For this example we are choosing VL and *CircuitFlow*.

In order to define a flow we need:

1. Starting node
2. Initializing node: usually set to 0
3. Circuit node: where we write the circuit we want lo launch
4. Repeat node: where the number of repetitions can be establish
5. End node
6. Links between each node we have created

[6]:
```
# Defining flow w/ CircuitFlow
flow = qsoa.CircuitFlow()
startNode = flow.startNode()
initNode = flow.initNode(0)
circuitNode = flow.circuitNode('Manual.Gates.Simon.QC_qSOA_Simon')
# Namespace + CircuitName
repeatNode = flow.repeatNode(1000)
endNode = flow.endNode()

flow.linkNodes(startNode, initNode)
flow.linkNodes(initNode, circuitNode)
flow.linkNodes(circuitNode, repeatNode)
flow.linkNodes(repeatNode, endNode)
print(flow.getFlowBody())
```

[6]:

```
{'class': 'go.GraphLinksModel', 'nodeDataArray': [{'category': 'Start', 'text':
'Start', 'key': -1, 'loc': ''}, {'category': 'Init', 'text': '0', 'key': -2,
'loc': ''},{'category': 'Circuit', 'text': 'Manual.Gates.Simon.QC_qSOA_Simon',
 'key': -3, 'loc': ''}, {'category': 'Repeat', 'text':
'1000', 'key': -4, 'loc': ''}, {'category': 'End', 'text': 'End', 'key': -5,
'loc': ''}], 'linkDataArray': [{'from': -1, 'to': -2, 'points': []}, {'from':
-2, 'to': -3, 'points': []}, {'from': -3, 'to': -4, 'points': []}, {'from': -4,
'to': -5, 'points': []}]}
```


2.3.2 Creating the flow

In order to create the flow we are using *createAssetFlowSync*, although the *createAssetSync* function would also work. The inputs that this function requires are:

- idSolution: to associate the flow with the solution we have selected before.

- assetName: to set the name of the flow.
- assetNamespace: to associate the flow with a class of flows that share something in common. In this case, we associate the flow with a set of basic flows.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the flow we have previously defined.
- assetLevel: to select either VL or IL, according to the definition of the flow.
- publish: to select if we want to publish the flow on qSOA® or not.

[7]:
```
## Flow creation
assetName = 'QF_qSOA_Simon'
assetNamespace = 'Manual.Gates.Simon'
assetDescription = 'Creating Simons flow from qSOA'
assetPublication = True

assetBody = flow
assetType = 'FLOW'
assetLevel = 'VL'

FlowManagementResult = qsoa.createAssetFlowSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetLevel,assetPublication)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAssetFlow* function creates, compiles and transpiles the flow, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAssetFlow*, and the *getAssetManagementResult* function.

2.4 Implementation of the algorithm

2.4.1 Executing the algorithm

We are now ready to execute the algorithm, so lets see on what platforms we can do so, and select the ones we are interested in.

[8]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
DeviceID = input("Select a device to run the flow in: ")
```

```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
Select a device to run the flow in: 14
```

Now, we proceed to run the quantum algorithm with the *runQuantumApplicationSync* function.

© QuantumPath® 2022 | All rights reserved

[9]:
```
exe_application = qsoa.runQuantumApplicationSync('NameTheTask', idSolution,␣
↪FlowID, DeviceID)
```

Note: Note that the synchronous version of this function is being used. This is because *runQuantumApplication* launches an execution, and the synchronous function waits for the execution to be done before moving on. If we do not know how long the execution is going to take (due to the execution itself or to the queue), or if it takes too long, the asynchronous version presents a clear advantage. However, for a better understanding of how this functions work, we recommend using the asynchronous version, *runQuantumApplication*.

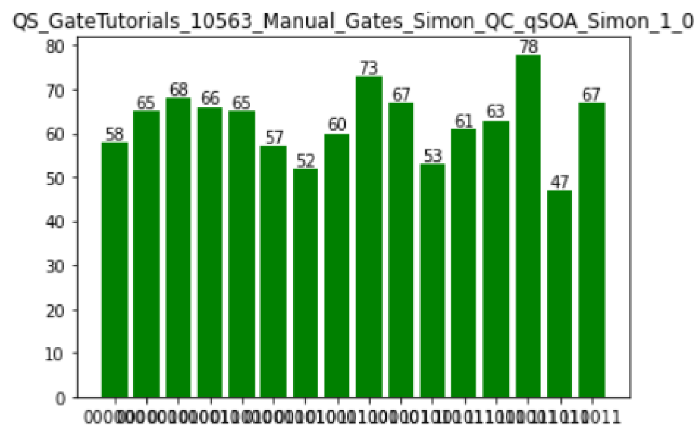
The function *runQuantumApplicationSync* gives an application object as output. For us to manage the results we need the *getQuantumExecutionResponse* function that returns the results as an execution object.

[10]:
```
restok_execution = qsoa.getQuantumExecutionResponse(exe_application.
↪getExecutionToken(), idSolution, FlowID)
restok_histogram = restok_execution.getHistogram()
print(restok_histogram)
```

```
{'QS_GateTutorials_12345_Manual_Gates_Simon_QC_qSOA_Simon_1_0': {'010011': 60,
'010010': 52, '101010': 53, '000011': 66, '111011': 67, '010000': 65, '000000':
58, '000010': 68, '101011': 61, '111010': 47, '010001': 57, '101000': 73,
'111001': 78, '101001': 67, '000001': 65, '111000': 63}}
```

Now that we have the results we asked for, we can proceed to represent them with the function *representResults*.

[11]:
```
# Circuit gate representation
gates_representation = qsoa.representResults(restok_execution)
```



© QuantumPath® 2022 | All rights reserved

2.4.2 Multiple devices

If we want to run the algorithm in multiple devices at the same time we can do so by creating an array with the information needed and proceeding the same way as before.

[12]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
```

```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
```

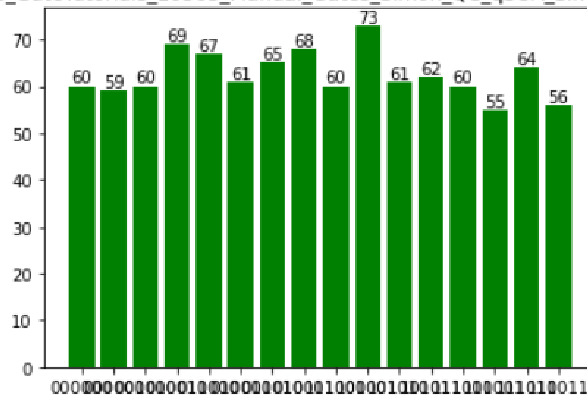
[13]:
```
## Run Quantum Gates Application
exe_ApplicationNames = ['Task_Amazonsim','Task_Qiskitsim']
exe_IdDevices = [14,2]

exe_Applications = [0] * len(exe_ApplicationNames)

for i in range(len(exe_ApplicationNames)):
    exe_Applications[i] = qsoa.runQuantumApplicationSync(exe_ApplicationNames[i],
    idSolution, FlowID, exe_IdDevices[i])
```

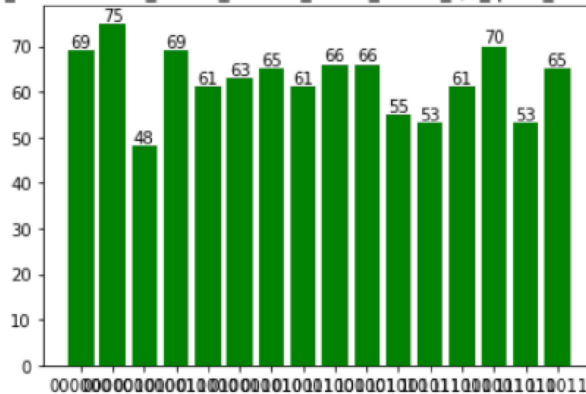
[14]:
```
# Get quantum execution response with execution token
restok_Executions = [0] * len(exe_ApplicationNames)
restok_Histograms = [0] * len(exe_ApplicationNames)

for i in range(len(exe_ApplicationNames)):
    restok_Executions[i] = qsoa.getQuantumExecutionResponse(exe_Applications[i].
 ↪getExecutionToken(), idSolution, FlowID)
    restok_Histograms[i] = restok_Executions[i].getHistogram()

print(restok_Histograms)
```

```
[{'QS_GateTutorials_12345_Manual_Gates_Simon_QC_qSOA_Simon_1_0': {'101011': 62,
'111011': 56, '111000': 60, '101001': 73, '111010': 64, '101000': 60, '010000':
67, '010011': 68, '010001': 61, '111001': 55, '010010': 65, '000010': 60,
'000001': 59, '000011': 69, '101010': 61, '000000': 60}},
{'QS_GateTutorials_12345_Manual_Gates_Simon_QC_qSOA_Simon_1_0': {'000000': 69,␣
 ↪'010001': 63, '010011': 61, '000010': 48, '101011': 53, '111000': 61, '101001':␣
 ↪66, '101010': 55, '000001': 75, '111011': 65, '010010': 65, '010000': 61,␣
 ↪'111001': 70, '000011': 69, '101000': 66, '111010': 53}}]
```

[15]:
```
# Circuit gate representation
for i in range(len(exe_ApplicationNames)):
    Gates_Representations = qsoa.representResults(restok_Executions[i])
```

© QuantumPath® 2022 | All rights reserved





We now proceed to analyze the results obtained in the simulation with just one device. It can be done analogously with the rest and the results should be equivalent.

The results shown in the histogram are the following:

© QuantumPath® 2022 | All rights reserved

Amazon Braket 25qbits Local Simulator		
Results	1st Register	Outcome
010011	010	60
010010	010	52
101010	101	53
000011	000	66
111011	111	67
010000	010	65
000000	000	58
000010	000	68
101011	101	61
111010	111	47
010001	010	57
101000	101	73
111001	111	78
101001	101	67
000001	000	65
111000	111	63

This means that the measurement of the first register can only be: $\{000, 010, 101, 111\}$.

Using these results we can figure out the value of s by solving the set of equations given by $s \cdot z = 0\ (mod 2)$.

For example, the measurement 010 tells us:

$$\begin{aligned} s \cdot 010 &= 0 \\ s_0 \cdot 0 + s_1 \cdot 1 + s_2 \cdot 0 &= 0 \\ \cancel{s_0 \cdot 0} + s_1 \cdot 1 + \cancel{s_2 \cdot 0} &= 0 \\ s_1 &= 0 \end{aligned}$$

From the 101 result we obtain the remaining parts:

$$\begin{aligned} s \cdot 101 &= 0 \\ s_0 \cdot 1 + s_1 \cdot 0 + s_2 \cdot 1 &= 0 \\ s_0 \cdot 1 + 0 \cdot 0 + s_2 \cdot 1 &= 0 \end{aligned}$$

Either $s_0 = s_2 = 0$, trivial solution, or $s_0 = s_2 = 1$. Therefore, the searched value is $s = 101$.

We can also verify that the other two results satisfy the equation: $s \cdot z = 0\ (mod 2)$, as it is easy to see that $101 \cdot 000 = 0\ (mod 2)$, and $101 \cdot 111 = 0\ (mod 2)$.

All being said, having figured out the value of the string s, we can corroborate that $f(x) = f(x \oplus s)$:

© QuantumPath® 2022 | All rights reserved

$$
\begin{aligned}
f(000) &= f(000 \oplus 101) = f(101) \\
f(001) &= f(001 \oplus 101) = f(100) \\
f(010) &= f(010 \oplus 101) = f(111) \\
f(011) &= f(011 \oplus 101) = f(110)
\end{aligned}
$$

References

[1] aQuantum, *QPath® Python SDK User Guide.* Available on QPath®.

© QuantumPath® 2022 | All rights reserved